Numerical assessment of non-electrolyte diffusiophoresis

Sergio Da Cunha[1], Natalia Shcherbakova[1], Vincent Gerbaud[1], Patrice Bacchin[1,*]

[1] Laboratoire de Génie Chimique, Université de Toulouse, CNRS, INP, UPS, Toulouse, France

* corresponding author: patrice.bacchin@univ-tlse3.fr

Supplementary Material:

C routines for numerical simulations of mixture flow

In this paper, diffusioosmosis and diffusiophoresis were investigated via numerical simulations. Section 5 in the paper mentioned that user-defined functions (UDFs) were necessary to capture the additional term $-Dn\nabla\Pi_{ic}$ in the solute transport equation, and to prescribe the motion of the sphere in the TEF model. For this extra convective term, three UDFs are required: one to allocate memory for Π_{ic}, one that assigns the values for Π_{ic} and another that calculates the scalar product of the overall advection field $\boldsymbol{\psi} = -D\nabla\Pi_{ic} + \boldsymbol{u}$ with the face normal vectors. These UDFs are given in Listing S1 and Listing S2 for the TFCV formulation in Section 4.2 in the paper.

Listing S1 – Allocating memory and assigning Π_{ic} values

```
/*********************************************************************
set_pi_ic.c contains two UDFs: an execute on loading UDF that reserves
one UDM for libudf and renames the UDM to enhance postprocessing,
and an on-demand UDF that sets the initial value of the UDM.
*********************************************************************/
#include "udf.h"
#define NUM_UDM 1
#define x_center 0.
#define y_center 0.
#define kic 100.
#define lic 1.e-7
#define att 0.
#define rad 2.e-7
#define diff_coeff 2.18e-10
/* diff_coeff in kg/m-s */
/* diff_coeff = D x rho */
static int udm_offset = UDM_UNRESERVED; /*makes sure udm_offset from loading is
                                          the same as udm_offest from demand*/
DEFINE_EXECUTE_ON_LOADING(allocate_udm_memory, libname)
```

```c
{
        if (udm_offset == UDM_UNRESERVED)
                udm_offset = Reserve_User_Memory_Vars(NUM_UDM);
        if (udm_offset == UDM_UNRESERVED)
                Message("Undefined UDM");
        else
        {
                Set_User_Memory_Name(udm_offset, "pi_ci_udm");
        }
        Message("\nUDM Offset for Current Loaded Library = %d", udm_offset);
}
DEFINE_ON_DEMAND(set_udm)
{
        Domain* d;
        Thread* ct;
        cell_t c;
        int i;
        real distance_cell_center;
        real xc[ND_ND];
        d = Get_Domain(1); /*Fluid domain; there's only one domain for the simulation*/
        if (udm_offset != UDM_UNRESERVED)
        {
                Message("Setting UDM\n");
                for (i = 0; i < NUM_UDM; i++)
                {
                        thread_loop_c(ct, d)
                        {
                                begin_c_loop(c, ct)
                                {
                                        C_CENTROID(xc, c, ct);
                                        distance_cell_center = sqrt(ND_SUM(pow(xc[0] -
                                          x_center, 2.), pow(xc[1] - y_center, 2.), 0.));
                                        C_UDMI(c, ct, udm_offset + i) = (1. + att) * kic *
                                          exp(-1. * (distance_cell_center - rad) / lic) -
                                          att * kic * exp(-1. * (distance_cell_center - rad)
                                          / (2. * lic));
                                        C_UDMI(c, ct, udm_offset + i) =
                                          C_UDMI(c, ct, udm_offset + i) * diff_coeff;
                                }
                                end_c_loop(c, ct)
                        }
                }
        }
        else
                Message("UDMs have not yet been reserved for current library\n");
}
```

Listing S2 – Setting solute advection field

```
/**********************************************************************/
/* UDF that implements a modified advective term in the */
/* scalar transport equation */
/**********************************************************************/
#include "udf.h"
#include "sg.h" /*to use INTERIOR_FACE_GEOMETRY*/
DEFINE_UDS_FLUX(set_convective_flow,f,t,i)
{
        real A[ND_ND], es[ND_ND], dr0[ND_ND], dr1[ND_ND], A_by_es, ds;
        cell_t c0, c1 = -1;
        Thread *t0, *t1 = NULL;
        real NV_VEC(psi_vec);
        real flux = 0.0;
        c0 = F_C0(f,t);
        t0 = F_C0_THREAD(f,t);
        F_AREA(A, f, t);
        real vx;
        /* If face lies at domain boundary, use face values; */
        /* If face lies IN the domain, use average of adjacent cells. */
        if (BOUNDARY_FACE_THREAD_P(t)) /*Most face values will be available*/
        {
                real dens;
                /* Depending on its BC, density may not be set on face thread*/
                if (NNULLP(THREAD_STORAGE(t, SV_DENSITY)))
                        dens = F_R(f, t); /* Set dens to face value if available */
                else
                        dens = C_R(c0, t0); /* else, set dens to cell value */
                NV_DS(psi_vec, =, F_U(f, t), F_V(f, t), F_W(f, t), *, dens);
                flux = NV_DOT(psi_vec, A); /* flux only comes from velocity */
                /*flux =0.;*/
        }
        else
        {

                c1 = F_C1(f,t); /* Get cell on other side of face */
                t1 = F_C1_THREAD(f,t);
                NV_DS(psi_vec, =, C_U(c0,t0), C_V(c0,t0), C_W(c0,t0), *, C_R(c0, t0));
                vx = C_U(c1, t1) - v0;
                NV_DS(psi_vec, +=, vx, C_V(c1, t1), C_W(c1, t1), *, C_R(c1, t1));
                flux = NV_DOT(psi_vec, A) / 2.0;
                INTERIOR_FACE_GEOMETRY(f, t, A, ds, es, A_by_es, dr0, dr1);
                flux = flux - A_by_es * (C_UDMI(c1, t1, 0) - C_UDMI(c0, t0, 0)) / ds;
                /* Flux from both velocity and Pi_ic */
        }
        return flux;
}
```

In Listing S1, DEFINE_EXECUTE_ON_LOADING is the UDF that allocates memory for the solute-interface interaction potential, and DEFINE_ON_DEMAND is the UDF that assigns its value. Variable C_UDMI in the second UDF corresponds to Π_{ic}, and its value is set in lines 54-59 according to eq. (4.4) in the paper. Note that the loops in the second UDF of Listing S1 guarantee that a value is assigned to all cells in all the cell zones of the domain.

In Listing S2, the advection field is calculated differently depending on whether the face is part of the boundaries (lines 20-31) or not (lines 32-44). The main reason for this distinction is that face values of $\nabla\Pi_{ic} \cdot \boldsymbol{A}$ (where $\boldsymbol{A}$ is the face area vector) are obtained via differencing between adjacent cells, which is not possible when the face is at a boundary. For boundary faces, it is simply assumed that $\nabla\Pi_{ic} = \mathbf{0}$. Note that this approximation is correct for the external boundaries of the domain, which are very far from the interface. However, $\nabla\Pi_{ic} = \mathbf{0}$ does not hold on the surface of the sphere. Still, this is not a real issue since the overall solute flux is set to 0 as a boundary condition for the interface.

Apart of the above mentioned UDFs, two other macros are necessary to implement the TEF model discussed in Section 4.1 in the paper. The first, named DEFINE_ADJUST, updates Π_{ic} at the beginning of every time step, since the structure of the cells change as the mesh deforms. Further, DEFINE_CG_MOTION is used to update the velocity of the sphere, according to the following expression: $v_{new} = v_{old} + F \times \Delta t$. In this expression, v^{new} is the updated velocity, v^{old} is the previous one, F is the resultant force along the axial direction, and Δt is the interval between two time steps. These macros are given in Listings S3 and S4.

Listing S3 – Updating Π_{ic} after remeshing

```
/*****************************************************************
update_pi_ic.c contains a DEFINE_ADJUST UDF that updates UDM at the
first iteration of every time step
*****************************************************************/
#include "udf.h"
#define NUM_UDM 1
#define y_center 0.
#define kic 100.
#define lic 1.e-7
#define rad 2.e-7
#define diff_coeff 2.18e-9
#define wall_id 14
/* diff_coeff in kg/m-s */
/* diff_coeff = D(2.18e-9) x rho */

DEFINE_ADJUST(update_udm,d)
{
        Dynamic_Thread* ndt = NULL;
        Thread* t1 = Lookup_Thread(d, wall_id);
        ndt = THREAD_DT(t1);
        real wall_center = DT_CG(ndt)[0];
        Thread* ct;
        cell_t c;
        int i;
        real distance_cell_center;
        real xc[ND_ND];
        real attrac;
        attrac = RP_Get_Input_Parameter("att");
        if (N_ITER % 50 == 0)
        {
                thread_loop_c(ct, d)
```

```
            {
                    begin_c_loop(c, ct)
                    {
                            C_CENTROID(xc, c, ct);
                            distance_cell_center =
                              sqrt(ND_SUM(pow(xc[0] - wall_center, 2.),
                              pow(xc[1] - y_center, 2.), 0.));
                            C_UDMI(c, ct, 0) = (1. + attrac) * kic *
                              exp(-1. * (distance_cell_center - rad) / lic) - attract
                              * kic * exp(-1. * (distance_cell_center - rad) /
                              (2. * lic));
                                    C_UDMI(c, ct, 0) = C_UDMI(c, ct, 0) * diff_coeff;
                    }
                    end_c_loop(c, ct)
            }
            Message("UDM updated, wall center = %.2e\n", wall_center);
    }
}
```

Listing S4 – Updating sphere velocity

```
/**********************************************************************
sets wall velocity
**********************************************************************/
#include "udf.h"
#define inlet_id 11
#define outlet_id 12
static real v_prev = 0.0;
#define mass 1.34e-12
DEFINE_CG_MOTION(wall_shift, dt, vel, omega, time, dtime)
{
        /*Mixture domain has id 1*/
        /*Find pointer to domain corps_surfacique*/
        Domain* dom;
        dom = Get_Domain(1);

        real wall_center;
        real force = 0.0;
        real dv;
        NV_S(vel, =, 0.0);
        NV_S(omega, =, 0.0);
        if (!Data_Valid_P())
                return;

#if !RP_HOST /*Serial or node*/
        /*Find thread corresponding to certain ID*/
        Thread* t1 = Lookup_Thread(dom, inlet_id);
        Thread* t2 = Lookup_Thread(dom, outlet_id);
        face_t f;
        real NV_VEC(A);
        real inforce = 0.0;
        real outforce = 0.0;

        begin_f_loop(f, t1)
        {
                F_AREA(A, f, t1);
                inforce += F_P(f, t1) * NV_MAG(A);
        }
        end_f_loop(f, t1)
```

```

            begin_f_loop(f, t2)
      {
            F_AREA(A, f, t2);
            outforce += F_P(f, t2) * NV_MAG(A);
      }
      end_f_loop(f, t2)

      #if RP_NODE /*Nodes only*/
            inforce = PRF_GRSUM1(inforce);
            outforce = PRF_GRSUM1(outforce);
      #endif

      force = (inforce - outforce) * 2. * M_PI;
#endif

      node_to_host_real_1(force); /*Now host and nodes know force*/

      dv = dtime * force / mass;
      v_prev += dv;
      wall_center = DT_CG(dt)[0];
      Message("time = %.2e, wall_vel = %.2e, force = %.2e, old_position = %.2e\n",
        time, v_prev, force, wall_center);
      vel[0] = v_prev;

}
```